Nama : Difa Bagasputra Maulana
NPM : 140810180057

Tugas 6 (Worksheet 6)

1. Adjacency Matrix

```
/*
Nama    : Difa Bagasputra Maulana
NPM     : 140810180057
Kelas   : A
Program : Adjacency Matrix
*/
#include <iostream>
using namespace std;

int vertArr[20][20];
int count = 0;

void displayMatrix(int v){
    int i, j;
    for (i = 1; i <= v; i++){
        for (j = 1; j <= v; j++)
        {
            cout << vertArr[i][j] << " ";
        }
        cout << endl;
    }
}

void add_edge(int u, int v){
    vertArr[u][v] = 1;
    vertArr[v][u] = 1;
}

int main(int argc, char *argv[]){
    int v=8;

    add_edge(1, 2);
    add_edge(1, 3);
    add_edge(2, 1);
    add_edge(2, 3);
    add_edge(2, 4);
    add_edge(2, 5);
    add_edge(3, 1);
    add_edge(3, 2);
    add_edge(3, 5);
    add_edge(3, 7);
    add_edge(3, 8);
    add_edge(4, 2);
    add_edge(4, 5);
    add_edge(5, 2);
    add_edge(5, 3);
    add_edge(5, 4);
    add_edge(5, 6);
    add_edge(6, 5);
    add_edge(7, 3);
    add_edge(7, 8);
    add_edge(8, 3);
```

```
    add_edge(8, 7);

    displayMatrix(v);
}
```

```
D:\Tugas\UNPAD\Analgo\Graf>g++ adj-matrix.cpp -o matriks

D:\Tugas\UNPAD\Analgo\Graf>matriks
0 1 1 0 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 0 1 0 1 1
0 1 0 0 1 0 0 0
0 1 1 1 0 1 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 1 0 0 0 1 0
```

2. Adjacency List

```
/*
Nama    : Difa Bagasputra Maulana
NPM     : 140810180057
Kelas   : A
Program : Adjacency List
*/
#include <iostream>
#include <cstdlib>
using namespace std;
/*
* Adjacency List Node
*/
struct AdjListNode{
    int dest;
    struct AdjListNode* next;
};

/*
* Adjacency List
*/
struct AdjList{
    struct AdjListNode *head;
};

/*
* Class Graph
*/
class Graph{
    private:
        int V;
        struct AdjList* array;
    public:
        Graph(int V){
            this->V = V;
            array = new AdjList [V];
            for (int i = 1; i <= V; ++i)
```

```
                array[i].head = NULL;
        }
        /*
        * Creating New Adjacency List Node
        */
        AdjListNode* newAdjListNode(int dest){
            AdjListNode* newNode = new AdjListNode;
            newNode->dest = dest;
            newNode->next = NULL;
            return newNode;
        }
        /*
        * Adding Edge to Graph
        */
        void addEdge(int src, int dest){
            AdjListNode* newNode = newAdjListNode(dest);
            newNode->next = array[src].head;
            array[src].head = newNode;
            newNode = newAdjListNode(src);
            newNode->next = array[dest].head;
            array[dest].head = newNode;
        }
        /*
        * Print the graph
        */
        void printGraph(){
            int v;
            for (v = 1; v <= V; ++v){
                AdjListNode* pCrawl = array[v].head;
                cout << "\n Adjacency list of vertex " << v << "\n head ";
                while (pCrawl){
                    cout<<"-> "<<pCrawl->dest;
                    pCrawl = pCrawl->next;
                }
                cout<<endl;
            }
        }
};


int main(){
    int n;
    cout << "Banyak node : "; cin >> n;
    Graph gh(n);
        gh.addEdge(1, 2);
        gh.addEdge(1, 3);
        gh.addEdge(2, 1);
        gh.addEdge(2, 3);
        gh.addEdge(2, 4);
        gh.addEdge(2, 5);
        gh.addEdge(3, 1);
        gh.addEdge(3, 2);
        gh.addEdge(3, 5);
        gh.addEdge(3, 7);
        gh.addEdge(3, 8);
        gh.addEdge(4, 2);
        gh.addEdge(4, 5);
        gh.addEdge(5, 2);
        gh.addEdge(5, 3);
        gh.addEdge(5, 4);
        gh.addEdge(5, 6);
```

```
        gh.addEdge(6, 5);
        gh.addEdge(7, 3);
        gh.addEdge(7, 8);
        gh.addEdge(8, 3);
        gh.addEdge(8, 7);

        gh.printGraph();

    return 0;
}
```

3. BFS

```
/*
Nama    : Difa Bagasputra Maulana
NPM     : 140810180057
Kelas   : A
Program : BFS
*/
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph{
    int V; // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V){
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w){
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::BFS(int s){
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;
```

```
    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty()){
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i){
            if (!visited[*i]){
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main(){
    // Create a graph given in the above diagram
    Graph g(8);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 3);
    g.addEdge(3, 7);
    g.addEdge(3, 8);
    g.addEdge(4, 5);
    g.addEdge(5, 3);
    g.addEdge(5, 6);
    g.addEdge(7, 8);

    cout << "Breadth First Traversal ";
    cout << "(mulai dari vertex 1) \n";
    g.BFS(1);

    return 0;
}
```

```
D:\Tugas\UNPAD\Analgo\Graf>g++ BFS.cpp -o bfs

D:\Tugas\UNPAD\Analgo\Graf>bfs
Breadth First Traversal (mulai dari vertex 1)
1 2 3 4 5 7 8
```

- BFS merupakan metode pencarian secara melebar sehingga mengunjungi node dari kiri ke kanan di level yang sama. Apabila semua node pada suatu level sudah dikunjungi semua, maka akan berpindah ke level selanjutnya. Dalam worst case BFS harus mempertimbangkan semua jalur (path) untuk semua node yang mungkin, maka nilai kompleksitas waktu dari BFS adalah O(|V| + |E|).

- Karena Big-O dari BFS adalah O(V+E) dimana V itu jumlah vertex dan E itu adalah jumlah edges maka Big-O = O(n) dimana n = v+e
- Maka dari itu Big-Θ nya adalah **Θ(n)**

4. DFS

```
/*
Nama    : Difa Bagasputra Maulana
NPM     : 140810180057
Kelas   : A
Program : DFS
*/
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph{
    int V; // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

Graph::Graph(int V){
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w){
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::DFSUtil(int v, bool visited[]){
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
```

```cpp
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v){
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

int main(){
    // Create a graph given in the above diagram
    Graph g(8);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 3);
    g.addEdge(3, 7);
    g.addEdge(3, 8);
    g.addEdge(4, 5);
    g.addEdge(5, 3);
    g.addEdge(5, 6);
    g.addEdge(7, 8);

    cout << "Depth First Traversal";
    cout << " (mulai dari vertex 1) \n";
    g.DFS(1);

    return 0;
}
```

```
D:\Tugas\UNPAD\Analgo\Graf>g++ DFS.cpp -o dfs

D:\Tugas\UNPAD\Analgo\Graf>dfs
Depth First Traversal (mulai dari vertex 1)
1 2 4 5 3 7 8
```

- DFS merupakan metode pencarian mendalam, yang mengunjungi semua node dari yang terkiri lalu geser ke kanan hingga semua node dikunjungi. Kompleksitas ruang algoritma DFS adalah O(bm), karena kita hanya hanya perlu menyimpan satu buah lintasan tunggal dari akar sampai daun, ditambah dengan simpul-simpul saudara kandungnya yang belum dikembangkan.
- Big O Kompleksitas total DFS () adalah (V+E). O(n) dengan V = Jumlah Verteks dan E = Jumlah Edges